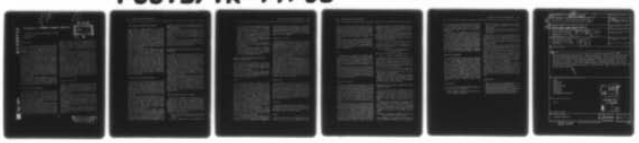
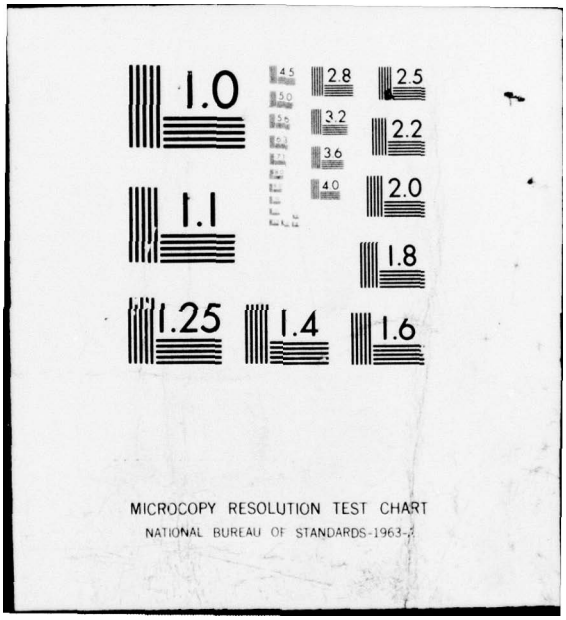
AD-A039 746 FEDERAL COBOL COMPILER TESTING SERVICE WASHINGTON D C F/G 9/2
EXPERIENCES IN COBOL COMPILER VALIDATION,(U)
MAY 77 G N BAIRD, M M COOK
UNCLASSIFIED FCCTS/TR-77/06 NL

| OF |
AD
A039 746



END
DATE
FILMED
6-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Experiences in COBOL compiler validation

by GEORGE N. BAIRD and MARGARET M. COOK

Department of the Navy
Washington, D.C.

INTRODUCTION

The Federal COBOL Compiler Testing Service (FCCTS) is an activity of the Software Development Division of the Department of the Navy, Automatic Data Processing Equipment Selection Office (ADPESO). Since July 1, 1972, all COBOL compilers brought into the Federal Government have to be identified as implementing one of the levels of the Federal COBOL Standard. The National Bureau of Standards, which has the responsibility for the development and maintenance of Federal ADP Standards, has delegated to the Department of Defense, and thereby to ADPESO, the responsibility for the operation of a Government-wide COBOL Compiler Testing Service. This responsibility is discharged by the FCCTS through the implementation and maintenance of the COBOL Compiler Validation System,[1] a comprehensive set of routines used to test COBOL compilers for compliance with the Federal COBOL Standard as prescribed in Federal Information Processing Standards Publication 21 (FIPS PUB 21),[2] published by the National Bureau of Standards.

This paper addresses several questions that arise in compiler validation. Why validate compilers for conformance to a standard? How is the validation performed? What experiences have been gained, and what conclusions can be derived from them? The questions will be discussed in turn.

WHY VALIDATE COMPILERS?

The purpose of validating a compiler is to ensure that syntactically correct programs compile and execute without abnormal termination, and that the semantics of the language being translated are correctly interpreted. Also, (where appropriate to the language), validation should point out the impact of implementor defined specifications which are allowed by the standard.

There are three phases in the computer systems acquisition cycle during which a validation is important. Prior to selection, a validation of the compilers for the various systems being proposed may constitute a part of the systems evaluation. After a computer system has been selected, a validation of the present compiler and a validation of the compiler to be procured disclose the areas of nonconformance in both compilers. The effort required in converting existing programs to the new system can then be realistically estimated prior to the changeover. After the delivery of a new computer system, but prior to acceptance, a compiler validation will reveal areas where a compiler does not meet the terms of the contract.

Our experience with the Validation System has shown that continuing benefits accrue from being aware of a compiler's status vis-a-vis its language standard. Compiler validation for computer systems which have already been acquired and are in present use can serve to point out which language elements do not operate correctly and therefore should not be used. A validation is particularly useful when a new version of a compiler or operating system is released, since it will immediately reveal errors in the revised software.

If a user has access to several different computer systems and is doing program development on all of these, he must know what language elements conform to the standard on each of the systems. Validation of the compilers on each system shows which language elements perform correctly. By writing programs using only these elements, a user ensures program portability.

SCOPE OF VALIDATION

Errors in compiling a program may arise from a single statement or a particular sequence of statements. Since validation verifies that individual language elements are processed correctly, errors in combining language elements may exist even though each of the separate elements are processed correctly.

A validation is not concerned with the efficiency of the object code generated, but only tests if the code is produced correctly. A validation system does not test implementor extensions to the language. If the implementor extensions cause problems in the standard language elements, a validation will identify these errors, but any errors in the use of the language extensions themselves will not be discovered during validation.

Finally, while a validation identifies problem areas in the use of standard language elements with a given compiler, it cannot indicate the ramifications of the compiler errors discovered.

FCCTS COBOL COMPILER VALIDATION

The validation of COBOL compilers by the Federal COBOL Compiler Testing Service is performed using the COBOL Compiler Validation System (CCVS), which was developed by the Department of the Navy Programming Languages Section under the direction of Capt. Grace M. Hopper, USNR. The CCVS consists of audit routines, their related data and an executive routine. Each audit routine is a COBOL program, and includes many tests of individual language elements. Supporting procedures indicating the results of the tests are included in each routine. The audit routines of the CCVS collectively contain the features of Federal Standard COBOL.

There are certain adjustments which must be made before the audit routines can be compiled upon a given computer system. First, names allowed by the COBOL standard to be implementor defined must be inserted into the audit routines before they can be compiled. Second, system control cards are required in order to compile and execute a COBOL program. File specification and allocation are also regulated by system control cards, and additional control cards are usually required by programs using the COBOL SORT verb. Third, the given system configuration may not include a hardware device or capability required by some of the test procedures in the CCVS, for example a system which does not support multiple unit assignment for mass storage devices. All references to multiple units must be deleted for the proper compilation of the audit routines on that system.

EXECUTING THE AUDIT ROUTINES

Input parameters to the CCVS executive routine[3] specify the implementor names, hardware dependent language elements to be deleted, and the operating system control cards required to compile the audit routines on a given computer system. The executive routine creates a file containing the audit routines with implementor names inserted in the proper place in the source code, and the operating system control cards required for compiling and executing each routine.

The audit routines in the CCVS consist of source code which is syntactically correct; the routines do not contain any tests which deliberately introduce incorrect syntax. Thus, each audit routine is expected to compile without errors. (This is frequently not the case. We have encountered "Standard" compilers where syntactically correct source code causes fatal diagnostic messages, compiler aborts, and even compiler loops.)

When an audit routine does not compile, or complete execution normally, the source code containing the language elements which the compiler could not handle is modified or deleted. The tests in the PROCEDURE DIVISION of the audit routines are coded so that a test is deleted by inserting NOTE at the beginning of the paragraph containing the test. This results in the entire paragraph being treated as a comment. The source paragraph following the deleted paragraph contains procedures which indicate the test has been deleted.

The CCVS executive routine contains an editing capability which permits addition, deletion, or replacement of source lines in the audit routines. After an audit routine has been modified so that it consists of only the language elements that the compiler accepts, the routines are again compiled and executed.

All of the supporting procedures for verifying whether a test passes or fails is contained in each routine. An output report is produced indicating the actual results of each test, and, when a test fails, the expected result.

ADDITIONAL INFORMATION FROM COMPILER VALIDATION

A compiler validation identifies many characteristics which can be used in comparing compilers. Due to compiler errors, valid syntax in the audit routines may be rejected or the resultant object program may abort during execution. The running of the CCVS on a system supplies information concerning the effectiveness of diagnostic messages. The effort required in locating the source code which caused execution to terminate abnormally can also be assessed.

The procedures used in validating a compiler give information on a system's ability to execute a program after fatal compilation errors have been diagnosed, or the effectiveness of a system supplied option for skipping execution if fatal errors are encountered.

Some minor programming aids may also be discovered during a validation. The audit routines can be compiled using options for cross reference listings of data-names and procedure-names. Some compilers flag blank cards. In some cases, we have uncovered hindrances. The suppression of the printing of the contents of columns 73 through 80 on a source listing is, for example, a hindrance to anyone running the CCVS, since the CCVS executive routine uses columns 73-80 to indicate whether a source line has been replaced or added.

THE VALIDATION SUMMARY REPORT

The output of a validation is a set of listings from the executive routine documenting the steps taken in preparing programs/jobs for execution; the compilation of each program; and the execution report of each program. These listings constitute the raw data from which a Validation Summary Report (VSR) is produced. (Any attempt to use the raw results for evaluating a compiler would be painful indeed due to the volume of paper involved.) The VSR provides the following information:

(a) The status of the compiler in relation to each of the four Federal levels of COBOL as defined in FIPS PUB 21.
(b) A list of language elements whose implementation

is not consistent with the language specification (American National Standard COBOL X3.23-1968).[4]

(c) A list of language elements which are not implemented due to a lack of hardware necessary to support those elements (e.g., read reversed, hardware switches, etc.). The language specification does not require the implementation of certain elements if they are dependent on specific hardware devices, and the system supporting the COBOL compiler itself does not support that device.

(d) Information-only items. These are necessitated due to the existence of imprecise language specifications in the Standard.

(e) Compiler characteristics noticed by the validation team. This includes the usefulness of diagnostic messages issued, format of the source program listing, and timings and memory requirements for the compiler and individual audit routines.

COBOL COMPILER PROBLEMS DISCOVERED DURING VALIDATIONS

The FCCTS has, since its establishment, validated compilers supplied by many different vendors. Many problem areas have been uncovered during these validations. Some of the problems are common to many compilers; others occurred only in particular ones.

In this section we present some of the common problem areas we have discovered in "Standard" COBOL compilers. The problems are grouped by the COBOL functional processing module in which they are found.

Nucleus

In a NOTE character string, any combination of characters from the computer's character set is treated as a comment; the string appears on the source listing, but is not compiled. From the NOTE tests included in the CCVS we have found that most COBOL compilers check the syntax of a NOTE statement. As an example, one of the tests is a NOTE sentence containing a single " (QUOTE) character. Most compilers generate a message indicating an illegal alphanumeric literal format was used since an alphanumeric literal is a character string enclosed in quotes. If the NOTE statement is not the first sentence of a paragraph, the NOTE comment ends with the first period followed by a space. On many systems though, the first period, with or without a trailing space, terminates the NOTE comment and attempts are made to compile the rest of the comment.

There are tests of the ADD and SUBTRACT CORRESPONDING statements with matching items requiring five levels of qualification. Most compilers give diagnostic error messages for these tests, but in some cases the compilation of the program was terminated.

A test of the floating insertion editing capability included in a Nucleus module audit routine moves 000123.45 to an elementary item with a PICTURE clause $$$B999.99. The language specification states that any of the simple insertion characters (comma, blank and zero) immediately to the right of floating insertion characters are part of the floating character-string. Thus, the contents of the edited item after the test should be $123.45. The result usually obtained for this test is $b123.45.

Sequential and random access

The size of the data records for a file may be specified by the RECORD CONTAINS integer-1 TO integer-2 CHARACTERS clause. The size of each record in the file is defined by the individual record descriptions and the RECORD CONTAINS clause is optional. If the clause is used, there should not be any restrictions in the File Description. We have found compilers which require the number of characters in each record to be a special item at the beginning of a record description when this clause is used. This is a nonstandard restriction.

One audit routine for both sequential and random access file processing includes a CLOSE file WITH LOCK statement. Attempts are then made to OPEN and READ the locked file during the current run unit. A file that has been closed with lock cannot be opened again during execution of the object program. Most of the systems tested abort the program execution with an error message stating that an attempt to access a locked file has been made. Some of the systems return data and continue executing as if the CLOSE WITH LOCK had not been encountered. In one case, the program went into an execution loop when an attempt to read a locked file was made.

Table handling

In the COBOL Table Handling Module, the OCCURS DEPENDING ON option specifies a table whose number of occurrences varies during execution. The number of items in the table during execution depends on the value of the data-name in the DEPENDING ON clause. If a SEARCH statement is encountered for a variable table, the last entry in the table is defined by the contents of the data-name. Some compilers accept the syntax for the table definition, but the table is always considered to be its maximum size in a SEARCH statement.

Segmentation

An independent program segment is expected to be in its initial state each time the segment is made available to the program. There are compilers which do not restore the initial state of independent program segments.

Library

The COPY statement with the REPLACING option allows the user to copy library text, replacing each occur-

rence of a word in the text with a new word. A variety of problems have been found when exercising this option in the audit routines. One compiler correctly handled a COPY REPLACING statement, but in a subsequent COPY without the REPLACING option of the same library text, words were replaced which should not have been. This caused undefined data-names during the program compilation.

Many compilers place restrictions on the REPLACING option which are not in the Standard. Some of these restrictions are that a qualified name could not be replaced; a data-name could not be replaced by a subscripted data-name; or a data-name could not be replaced in an 01 level entry in the Data Division.

Sort

A frequent restriction encountered in the SORT module is a limitation in the specification of the minimum number of characters in a sort record description. Usually the compilation of the COBOL source program will not cause any diagnostic messages, but when the sort program is executed, a message indicating incorrect record length is produced.

Problems have arisen when one of the sort keys is defined as a signed numeric field. For a sort on ascending keys, negative values should appear before positive values. In some cases, signed numeric items have not been sorted in the correct order because comparison was based on the actual binary structure of the data, and not the algebraic value associated with the data.

Implementation variations

A file may contain multiple record descriptions, with each record having a different length. It is important for a user to know how much external storage media must be allocated for records of a given file. In order to compute this a user must know if a given implementation always writes records of the maximum length, or if variable length records are written. This would be especially significant if most of the records in the file were much shorter than the remainder. Though not required by the standard, a great deal of external storage space is wasted if fixed length records are written.

There are procedures in both sequential and random access audit routines which create a file containing variable length records, and in the subsequent reading of the file test if the system creates all records as the maximum fixed length.

As a result of our validations, we have changed some tests to information-only tests because of language ambiguities. For example, there is no explicit statement in the Standard as to what the result should be when moving a signed numeric field to an alphanumeric field. There are statements suggesting that any appropriate conversion takes place during an elementary move, but there could be doubt as to whether the elimination of a sign is included. In order to determine the result for a particular compiler, the alphanumeric item is tested for a numeric value after moving a +1 and a −1 to the alphanumeric item.

The use of optional words is only for readability and they are not supposed to effect the meaning of the statements. However, we have found one case where the presence of the optional word 'IS' changes the meaning of a statement. One of the audit routine tests is

IF A = B AND IS NOT GREATER C OR D.

The presence of the word 'IS' leaves no doubt that NOT is part of the relational operator NOT GREATER. As a result, the expansion gives

IF A = B AND A IS NOT GREATER C OR *A IS NOT GREATER D.*

But for the combined abbreviated relational condition:

IF A = B AND NOT GREATER C OR D,

the language specification states explicitly that the word NOT is part of the logical connector AND NOT, and is not part of the relational operator NOT GREATER. The effect of the rule causes the statement to be expanded so that the NOT is associated with the operand called C, and not associated with the operand called D:

IF A = B AND NOT A GREATER C OR *A GREATER D.*

The presence of the word 'IS' in the statement can indeed violate the law of least astonishment.

The discovery of problems in tests such as those described above has resulted in recommendations for language clarification to the American National Standards Institute Technical Committee X3J4, which has the responsibility for the maintenance of X3.23-1968 COBOL.

REASONS FOR FAILURES IN EXECUTING THE AUDIT ROUTINES

There are several reasons why a compiler may be implemented in such a way that there are discrepancies between the language specifications and the results given by the compiler. The most common reason for discrepancies is simply due to logic errors in the compiler. This can be attributed to lack of adequate controls in producing compilers.

A second reason for differences in the implementation is the misinterpretation of the language specification on the part of the implementor. One case in point (which has been noticed in several compilers) is the use of the word THRU in the current standard (X3.23-1968). In many specifications throughout the document, the word THRU appears as follows:

PERFORM procedure-name-1 THRU procedure-name-2
FILE-LIMIT literal-1 THRU literal-2, etc.

There is nothing in the above syntax which indicates that the words THRU and THROUGH are interchangeable.

The only reference that establishes this little known fact is the reserved word list where they are shown to be equivalent. This probably accounts for the fact that we have identified several compilers which do not allow usage of THROUGH.

Another problem is the ambiguity that is inherent in a language as complex as COBOL. These are areas in the current language specification that, at best, are ill-defined, and the implementor must make a unilateral decision as to the direction the implementation will take. A good example would be the default action the compiler takes for a WRITE statement to the printer, when the ADVANCING phrase is not specified. Based on whether the default assumes WRITE BEFORE ADVANCING or WRITE AFTER ADVANCING inappropriate spacing or overprinting of lines can occur. This was recognized as a shortcoming of the language, and subsequently corrected so that a default is now specified.

RESOURCES REQUIRED

An accounting summary is prepared for each validation performed. This summary includes professional personnel time for required modifications to the CCVS (to accommodate any compiler peculiarities), site visit for acquisition of raw data, evaluation of raw data, and preparation of the VSR; and processor time required for executing the audit routines. An average validation has thus far required 73 hours of professional time and 29 hours of clerical time. The processor time will naturally vary with the computer used. Execution of all the audit routines takes approximately 20 minutes of UNIVAC 1108 processor time. We estimate the average cost of a validation to be approximately $1,000, although these figures are subject to wide variations.

CONCLUSIONS

The use of the CCVS in validating COBOL compilers attempts to answer three major questions:

- Will the compiler accept the syntax as defined in the COBOL language specification?
- Will the compiler generate the appropriate object code to satisfy the semantic requirements of the language specifications?
- What unilaterial actions does the compiler take when the language specification leaves the result up to the implementor?

The results of over a year's work in validating a variety of compilers indicate that there may not be a compiler which completely conforms to the COBOL standard. In a few cases we were tempted to question whether the compiler was in fact compiling COBOL, or some other language similar to COBOL!

The reasons for the amount of non-conformance or deviation from the language specification can be blamed partly on the 1968 COBOL Standard. Most of these problem areas have been resolved during the development of the revision to X3.23-1968. As a result, we feel that we can expect to see better compilers since the language specifications are tighter and better defined; the idea of providing standard compilers is being encouraged in the marketplace by the users; and, most importantly, we have a measurement tool which can be used to determine the degree to which a compiler conforms to the Standard.

We recognize that the Validation System is necessarily incomplete. But we also are convinced of the importance of having some capability for measuring the quality of software. What we have learned during the period we have been validating compilers confirms the importance of software engineering, and thereby the importance of any measurement tool which results in software quality improvement.

REFERENCES

1. Baird, G. N., "The DOD Compiler Validation System," *Proc. 1972 FJCC*, AFIPS Press, Volume 41, Pages 819-827.
2. *Federal Information Processing Standards Publication 21*, U.S. Government Printing Office, Washington, D.C., March 1972.
3. *Navy COBOL Compiler Validation System User's Guide*, Information Systems Division, Department of the Navy, January 1973.
4. *American National Standard COBOL X3.23-1968*, American National Standards Institute Incorporated, New York 1968.

BIBLIOGRAPHIC DATA SHEET	1. Report No. (14) FCCTS/TR-77/06	2.	3. Recipient's Accession No.
4. Title and Subtitle (6) Experiences in COBOL Compiler Validation			5. Report Date 9 May 1977
			6.
7. Author(s) (10) George N. Baird and Margaret M. Cook			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Federal COBOL Compiler Validation Service ADPE Selection Office Department of the Navy Washington, D. C. 20376 (11) 9 May 77			10. Project/Task/Work Unit No.
			11. Contract/Grant No.
12. Sponsoring Organization Name and Address ADPE Selection Office Department of the Navy Washington, D. C. 20376 (12) 6p.			13. Type of Report & Period Covered
			14.

15. Supplementary Notes

16. Abstracts

This technical paper goes into the subject of software verification, COBOL compiler validation in particular. This research is a result of the work performed by the Federal COBOL Compiler Testing Service (FCCTS). This organization is the only one of its kind in the Federal Government in that it deals with the quality assurance of software. The paper discusses why to validate software, the scope of software validation as well as the COBOL Compiler Validation System (CCVS) used by the FCCTS in performing its mission of validating all COBOL compilers brought into the Federal inventory.

17. Key Words and Document Analysis. 17a. Descriptors

COBOL
Validation
Software
Audit Routines
Verifying
Compilers
Standards
Programming Languages

17b. Identifiers/Open-Ended Terms

CCVS

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release unlimited.	UNCLASSIFIED	5
	20. Security Class (This Page) UNCLASSIFIED	22. Price